A Smalltalk Virtual Machine Architectural Model

Allen Wirfs-Brock
Pat Caudill
Instantiations, Inc.

Allen_Wirfs-Brock@Instantiations.com
Pat_Caudill@Instantiations.com

This document describes a bytecode architecture for an experimental Smalltalk virtual machine. The architecture was originally developed by the authors at Digitalk Inc. in 1993-1994. This document is published with the permission of ObjectShare Inc, the successor company to Digitalk.

This architecture is intended to be used for the representation of compiled methods in the context of a dynamic translating ("JIT") virtual machine. As such, its main purpose is to provide information that can be easily and efficiently processed by such a translator. An important design goal was the minimization of the space needed to represent compiled methods. Because it was designed to be the input to a translator, minimization of decoding time was not a consideration of the design. It was not intended for direct interpretation and certain features, such as the "Label" construct would be inefficient if used by an interpreter.

Architectural influences include the Smalltalk-80 virtual machine, Tektronix Smalltalk implementations, and various Digitalk ("Smalltalk/V") virtual machines. The basic architecture is that of an accumulator+stack machine. Unlike the Smalltalk-80 virtual machine but similar to the Digitalk vm's, the architecture uses an explicit stack of activation records instead of a linked-list of context objects. Most arguments and local variables are allocated in an activation record but an activation record may also reference a heap allocated "context" object (in retrospect, "environments" would have been a better name for these objects) that contain any local variables that have non-FIFO lifetimes. The bytecode compiler is responsible for determining which methods require context objects and for determining which variables need to be allocated in contexts.

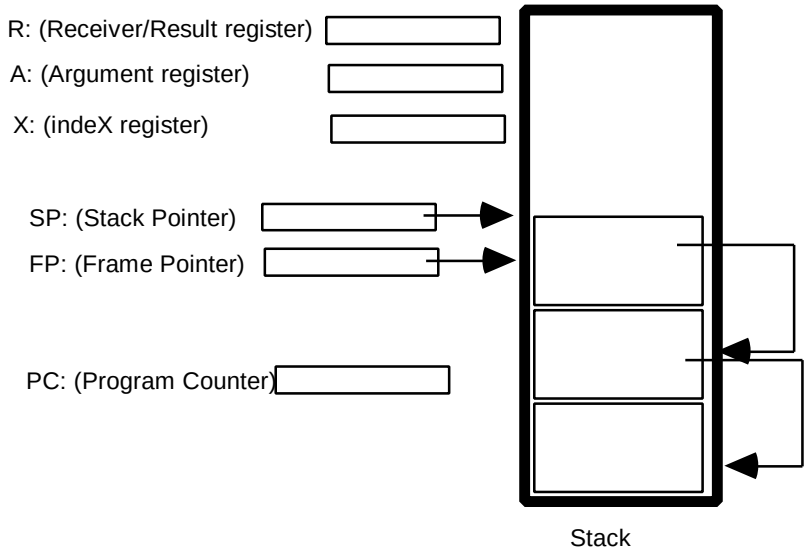
The register+stack architecture was driven by these observations about Smalltalk programs:

- *Deep call chains*
- *Frequent adjacent calls*
- *Most computation happens in primitives and leaf methods*
- *Most methods have 0 or 1 arguments (plus receiver)*

The registers ensure that the receiver and arguments arrive at primitives or leaf methods in machine registers. Such methods typically do not need full activation records and do not need to store the arguments to the stack. Non-leaf methods save their arguments to the stack, where they stay across sequences of adjacent calls.

The X (index) register is used for indirect address and is primarily used for up-level addressing of lexically nested blocks.

Base Architectural Elements



The execution model of the virtual machine consists of a *Stack* and several special purpose registers. The stack consists of a sequence of OOPS, the last of which is pointed to by the *SP* register. Individual OOPS may be added or removed from the stack by incrementing or decrementing the SP. Logically, the stack is organized into a sequence of *Stack Frames (*or activation records) that represent the state of individual procedure activations. The newest frame is pointed to by the *FP* register. Each frame contains a back reference to the preceding frame.

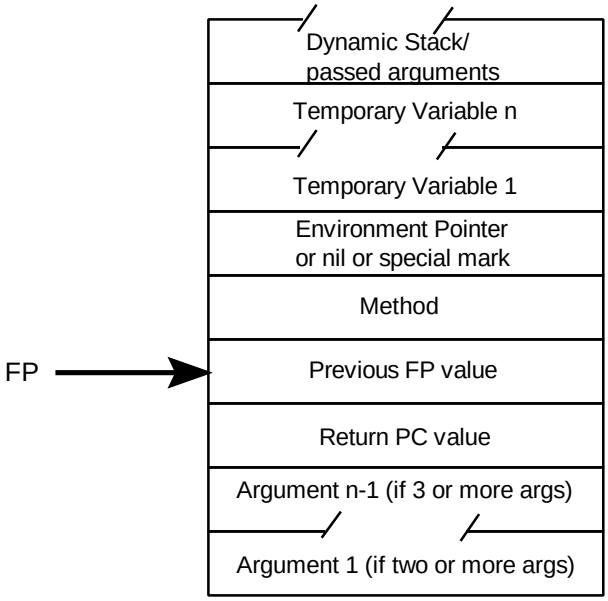
The PC register contains a value that identifies the next instruction to be executed.

The *R* register contains the OOP of the object that is the receiver of a message send instruction. Upon return from a message send, it contains the value returned by the procedure.

The *A* register is used to pass the last (leftmost) argument to a procedure. Other arguments are passed on the stack. The A register is volatile across procedure calls. Upon return from a call its contents are undefined.

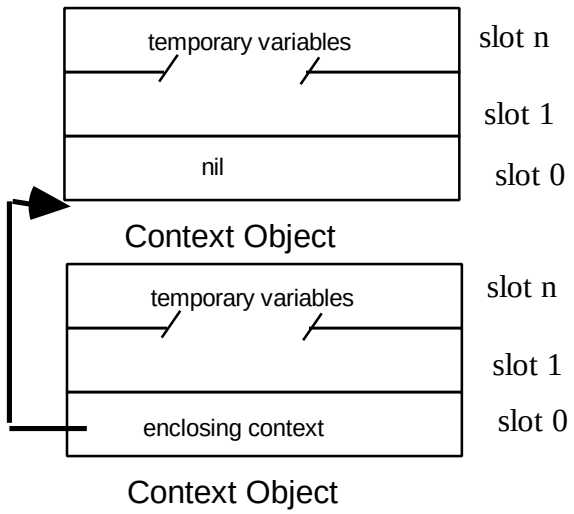
The *X* register is used as an index or base register to indirectly addess the fields of objects. Certain instructions that perform explicit indirect access through stack-based values also implicitly load the X register. The X register is volatile across procedure calls. Upon return from a call its contents are undefined.

Stack Frame Format



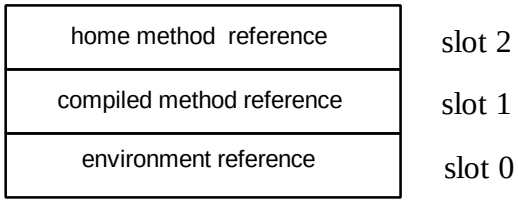
Contexts

A stack frame may reference an off-stack *context* through its environment pointer. A context is used to contain the storage for any local variable that may be referenced by nested blocks or which may need to be retained after the stack frame is discarded. Contexts may be logically nested, the first field of a context contains a reference to its enclosing context or nil if there is not an enclosing context. A stack frame for a method must have a context if the method contains any blocks with non-local (up-arrow) returns.



Blocks

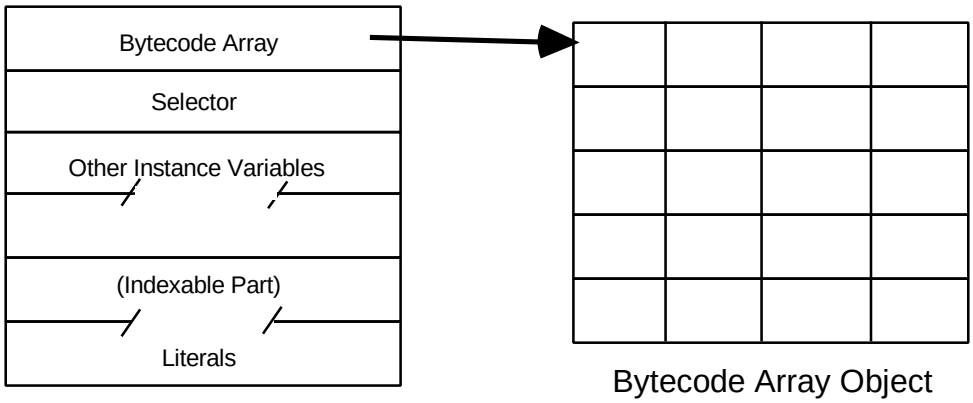
A *block*[1] is an object that when sent the message value (or one of its variants) executes a block of code in some pre-specified environment. A block object is created each time a literal code block (code enclosed within square brackets) is encountered during program execution. The block object contains a copy of the environment pointer of the stack frame that was active when the block was created and a reference to a compiled method that contains the executable code for the block (Each literal block is compiled as an independent compiled method).



Block Object

Compiled Methods

A Compiled Method is an object that encodes the executable form of a method or block. It consists of a set of bytecodes that encode the actual statements to be executed and a set of literals which are references to objects used by the code. A Compiled Method is actually a composite object structure of the following form:



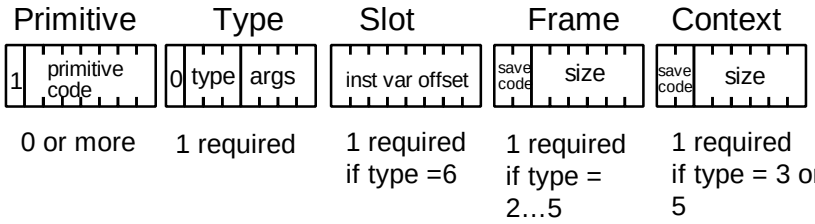
[1]Block Closure would be a technically more correct name, but is probably less descriptive to the Smalltalk programmer. Intuitively, such a programmer would expect the value of the variable x, after execution of the statement:

x := [...].

to be a "block" .

The majority of the information about a compiled method is encoded in its bytecode array. The first few bytes (at least 1) is a declarative header that describes the execution environment required for the method. This is normally followed by actual bytecodes that describe the imperative actions to be performed by the methods. In a few situations, the header is sufficient to describe the method and no bytecodes are required.

Bytecode Header



Type	Meaning	Argument
1	leaf	number of arguments
2	simple method	number of arguments
3	method w/context	number of arguments
4	simple block	number of arguments
5	block w/context	number of arguments
6	accesser/setter	0=accessor, 1=setter
0,7	reserved	

Save Code	Meaning
0	Don't save any regist
1	Save R in first temp s
2	Save A in first temp s
3	Save R in 1st temp s and A in 2nd temp s

Primitive Byte

If a method invokes a virtual machine primitive, the first byte of the bytecode header is a primitive byte. A primitive byte is identified by its high order bit being set to one. The low order 7 bits of a primitive byte encode the primitive number. A bytecode header may begin with several primitive bytes. In case the primitive number is formed by concatenating in left-to-right order the low order 7 bits of each primitive byte. A single primitive byte can encode primitive numbers in the range 0-127. Two bytes are need to encode primitive in the range 128-16384. The encoding for larger primitive numbers is defined but not currently supported by the virtual machine.

Type Byte

Every bytecode header contains a type byte. If the method has no primitive bytes then the type byte will be the first byte of the header. If there are primitive bytes, the type byte is the first byte following the last primitive byte. A type byte is distinguished from a primitive byte by having its most significant bit set to zero.

The three bit type field identifies the execution environment required for the method. It also defines the interpretation of the argument field and of any additional header bytes.

Method Types

Type 1 – Leaf Methods and Blocks
Type code 1 identifies a method that is a leaf method and does not need to have an activation record created for it. The argument field specifies the number of arguments that are passed to the method. A method is a leaf method if it contains no message sends or other operations that could result in the activation of another method or block and does not require any temporary variables. In addition a Leaf Method may be used to represent a block if the block does not make use of its environment reference. This is only the case when the block does not reference any variables defined outside the block or does not contain any out-of-scope returns.

Type 2 – Simple Method Frame
Type code 2 identifies a method (not a block) that requires a simple activation record. That is, an activation record that does not include a heap allocated context (environment). The environment pointer field of the activation record will be set to nil. The argument field specifies the number of arguments that are passed to the method. The type byte is immediately followed by a Frame Byte. The lower 6 bits specify how many temporary variables should be allocated within the activation record. All temporary variables are initialized to nil except the first two which may be initialized to the values of the R and A registers depending upon the setting of the high order two bits of the Frame Byte.

Type 3 – Method Frame with Context
Type code 3 identifies a method (not a block) that requires an activation record that includes a heap allocated context (environment). The environment pointer field of the activation record references the context which is allocated as an instance of the class identified in slot TBD of the known object table. The argument field specifies the number of arguments that are passed to the method. The type byte is immediately followed by a Frame Byte. The lower 6 bits specify how many temporary variables should be allocated within the activation record. All temporary variables are initialized to nil except the first two which may be initialized to the values of the R and A registers depending upon the setting of the high order two bits of the Frame Byte. The Frame Byte is immediately followed by a Context Byte. The lower 6 bits specify how many temporary variables should be allocated within the context. All temporary variables are initialized to nil except the second and third which may be initialized to the values of the R and A registers depending upon the setting of the high order two bits of the Context Byte. Note that the first field of the context (the enclosing environment reference) is set to nil.

Type 4 – Simple Block Frame
Type code 4 identifies a block (not a method) that requires a simple activation record. That is, an activation record that does not include a heap allocated context. The block object is referenced by register R. The environment pointer field of the activation record is set to the value in the environment reference field of the block. The argument field specifies the number of arguments that are passed to the block. The type byte is immediately followed by a Frame Byte. The lower 6 bits specify how many temporary variables should be allocated within the activation record. All temporary variables are initialized to nil except the first two which may be initialized to the values of the R and A registers depending upon the setting of the high order two bits of the Frame Byte.

Type 5 – Block Frame with Context
Type code 3 identifies a block (not a method) that requires an activation record that includes a heap allocated context. The block object is referenced by register R. The environment pointer field of the activation record references the context which is allocated as an instance of the class identified in slot TBD of the known object table. The enclosing environment field of the activation record is set to the value in the environment reference field of the block. The argument field specifies the number of arguments that are passed to the method. The type byte is immediately followed by a Frame Byte. The lower 6 bits specify how many temporary variables should be allocated within the activation record. All temporary variables are initialized to nil except the first two which may be initialized to the values of the R and A registers depending upon the setting of the high order two bits of the Frame Byte. The Frame Byte is immediately followed a Context Byte. The lower 6 bits specify how many temporary variables should be allocated within the context. All temporary variables are initialized to nil except the second and third which may be initialized to the values of the R and A registers depending upon the setting of the high order two bits of the Context Byte.

Type 6 – Accessor/Setter Methods
Type code 6 identifies a method whose only action is to immediately return the value of one of the receiver's instance variables (a accessor) or to immediately assign the value of the A register to one of the receiver's instance variables (a setter). The argument field specifies the number of arguments that are passed to the method and whether this is an accessor or a setter. A value of 0 indicates a accessor while a value of 1 indicates a setter. The offset of the instance variable within the receiver is specified in the byte that immediately follows the type byte. Type 6 methods do not require the creation of an activation record.

Bytecodes

Label n	Define label n
(0=n=7)	Code 00+n
(8=n= 39)	Code F0:00+n-8
(40=n= 8231)	Code F0:20+(n-39>>8):(n-39&ff)

Identifes a branch point within the method which is the target of one or more conditional or unconditional jump instructions.

Jump n	Unconditional jump to label n
(0=n=7)	Code 08+n
(8=n= 39)	Code F0:40+n-8
(40=n= 8231)	Code F0:60+(n-39>>8):(n-39&ff)
JumpT n	Conditional jump to label n if true
(0=n=7)	Code 10+n
(8=n= 39)	Code F0:80+n-8
(40=n= 8231)	Code F0:A0+(n-39>>8):(n-39&ff)
JumpF n	Conditional jump to label n if false
(0=n=7)	Code 18+n
(8=n= 39)	Code F0:C0+n-8
(40=n= 8231)	Code F0:E0+(n-39>>8):(n-39&ff)

Control is either conditionally or unconditionally transfered to the bytecode immediately following the Label or XLabel bytecode identified by the parameter n. JumpT and XJumpT transfers control if the R register contains the value true. JumpF and XJumpF transfers control if the R register contains the value false. If the label target of the jump is earlier in the method (a backwards branch) a virtual machine interrupt may occur.

LATmp n	Load A register from stack temporary n
(0=n=7)	Code 20+n
(0=n= 63)	Code F1:00+n
LRTmp n	Load R register from stack temporary n
(0=n=7)	Code 28+n
(0=n= 63)	Code F1:80+n

The value contained in temporary slot n of the current activation record is loaded into the A or R register.

LALit n	Load A register from literal n
(0=n=7)	Code 30+n
(8=n=255)	Code F2:00+n
LRLit n	Load R register from literal n
(0=n=7)	Code 38+n
(8=n= 255)	Code F3:00+n

The value contained in literal slot n of the current method is loaded into the A or R register. Literal slot 0 is the first indexable field of the literal array, slot 1 is the second field, etc.

LAAsc n	Load A register from association in literal n
(0=n=7)	Code 40+n
(8=n=255)	Code F4:00+n
LRAsc n	Load R register from association in literal n
(0=n=7)	Code 48+n
(8=n= 255)	Code F5:00+n

The 2nd instance variable (slot 1) of the object contained in literal slot n of the current method is loaded into the A or R register. Literal slot 0 is the first indexable field of the literal array, slot 1 is the second field, etc. The object stored in the literal is normally an Association.

LAEnv n	Load A indirect through environment pointer
(0=n=7)	Code 50+n
(0=n= 63)	Code F6:00+n
LREnv n	Load R indirect through environment pointer
(0=n=7)	Code 58+n
(0=n= 63)	Code F6:40+n

The environment pointer from the active stack frame is loaded into the X register. The value contained in slot n of the context object referenced by the environment pointer from the active stack frame is loaded into the A or R register.

LAT0 n	Load A register indirect through temporary 0
(0=n=7)	Code 60+n
(0=n= 255)	Code F7:00+n
LRT0 n	Load R register indirect through temporary 0
(0=n=7)	Code 68+n
(0=n= 255)	Code F8:00+n

The value of temporary variable 0 of the active stack frame is loaded into the X register. The value contained in slot n of the object referenced by temporary variable 0 of the active stack frame is loaded into the A or R register.

LAI n	Load A register indirect through X register
(0=n=3)	Code 70+n
(4=n= 67)	Code F6:80+n-4
(4=n= 255)	Code F6:BF:n
LRI n	Load R register indirect through X register
(0=n=3)	Code 74+n
(4=n= 67)	Code F6:C0+n-4
(4=n= 255)	Code F6:FF:n

The value contained in slot n of the object referenced by register X is loaded into the A or R register.

LXTmp n	Load X register from stack temporary n
(0=n=7)	Code E3:20+n
(0=n= 63)	Code E3:F1:00+n
LXLit n	Load X register from literal n
(0=n=7)	Code E3:30+n
(8=n=255)	Code E3:F2:00+n
LXAsc n	Load X register from association in literal n
(0=n=7)	Code E3:40+n
(8=n=255)	Code E3:F4:00+n
LXEnv n	Load X register from environment variable n
(0=n=7)	Code E3:50+n
(8=n=255)	Code E3:F6:00+n
LXT0 n	Load X register from environment variable n
(0=n=7)	Code E3:60+n
(8=n=255)	Code E3:F7:00+n
LXI n	Load X register indirect through X register
(0=n=3)	Code E3:70+n
(4=n= 66)	Code E3:F6:80+n-4
(4=n= 255)	Code E3:F6:BF:n

The the designate value is loaded into the X register. Note that the encoding of these instructions consist of the ALT byte 16rE3 followed by the instruction encoding of the corresponding instruction to load the A register.

LAArg n	Load A register from stacked argument n
(0=n=1)	Code 94+n
(2=n= 14)	Code F9:A0+n
LRArg n	Load R register from stacked argument n
(0=n=1)	Code 96+n
(2=n= 14)	Code F9:B0+n

The value contained in stacked argument slot n of the active stack frame is loaded into the A or R register. Slot 0 corresponds to the second to last (second

to leftmost) argument of a method with two or more arguments (the leftmost argument is always passed in register A). For a method with m (2=m=15)) arguments the stacked argument slot for the i'th argument from the right (1=i<m) is m-i-1.

LAconst x	Load the designated constant into the A register
(x=nil)	Code EA
(x=true)	Code EC
(x=false)	Code EE
(x=0)	Code D8
(x=1)	Code D9
(x=2)	Code DA
(x=n, -32=n=31)	Code F9:C0+n (encoded as 6-bit excess-32 integer)
LRconst x	Load the designated constant into the R register
(x=nil)	Code EB
(x=true)	Code ED
(x=false)	Code EF
(x=0)	Code E3:D8
(x=1)	Code E3:D9
(x=2)	Code E3:DA
(x=n, -32=n=31)	Code E3:F9:C0+n (encoded as 6-bit excess-32 integer)

The designated constant is loaded into either the A or R register. Nil, true, and false are the values contained in slots TDB of the known object table..

LRStk n	Load R register from the stack
(0=n=2)	Code DC+n
(0=n= 5)	Code F9:80+n

The value contained on the stack in the n'th slot below the top of stack is loaded into the R register. n=0 load the value currently on the top of the stack.

SRTmp n	Store R register into stack temporary n
(0=n=7)	Code 78+n
(0=n= 63)	Code F9:00+n

The value contained in register R is stored in temporary slot n of the current activation record.

SATmp n	Store A register into stack temporary n
(0=n=7)	Code E378+n
(0=n= 63)	Code E3F9:00+n

The value contained in register A is stored in temporary slot n of the current activation record.

SREnv n	Store R register indirect through environment pointer
(0=n=7)	Code 80+n
(0=n= 63)	Code F9:40+n

The value contained in register R is stored in slot n of the context object referenced by the environment pointer of the active stack frame.

SAEnv n	Store A register indirect through environment pointer
(0=n=7)	Code E3:80+n
(0=n= 63)	Code E3:F9:40+n

The value contained in register A is stored in slot n of the context object referenced by the environment pointer of the active stack frame.

ST0 n	Store R register indirect through temporary 0
(0=n=7)	Code 88+n
(0=n= 255)	Code FA:00+n

The value contained in register R is stored into slot n of the object referenced by temporary variable 0 of the active stack frame.

SI n	Store R register indirect through X register
(0=n=3)	Code 90+n
(0=n= 255)	Code FB:00+n

The value contained in register R is stored into slot n of the object referenced by register X.

RtoA	Move R register to A register
	Code E8
AtoR	Move A register to R register
	Code E3:E8

The value contained in register R is copied into the A register. The R register is not modified.

PushR	Push R register onto stack
	Code E6

The value contained in register R is pushed onto the stack. The R register is not modified.

PopA	Pop A register from stack
	Code E4
PopR	Pop R register from stack
	Code E5

The value currently contained in the top element of the stack is placed into the designated register. The element is removed from the stack.

DeleteTOS	Delete top stack element
	Code E7

The value contained in top element of the stack is removed from the stack and discarded

BlockA	Create a Block object in the R register
	Code E3:E9
BlockR	Create a Block object in the A register
	Code E9

A new block object is created. The current value of the designated register is placed into the Method slot of the block and the environment pointer from the current stack frame is placed in the environment slot of the block. The Class of blocks is contained in known object slot TBD. A reference to the newly created block is left in the designated register.

SendC n	Send a message with a common selector
(0=n=39)	Code B0+n
SendLit x,n	Send a x argument message with a literal selector
(0=n=11,0=x=1)	Code 98+(n*2)+x
(12=n=43,x=0)	Code FC:00+(n-12)
(12=n=43,x=1)	Code FC:20+(n-12)
(0=n=31,2=x=4)	Code FC:((x-1)<<6)+n
(0=n=2047,0=x=15)	Code FD:(x<<3)+(n>>8):n&FF
LRSendDrop x,n	Load R, Send a x argument message, Delete TOS
(0=n=31,2=x=4)	Code FC:((x-1)<<6)+20+n
(0=n=2047,0=x=15)	Code FD:80+(x<<3)+(n>>8):n&FF

The message lookup algorithm using the designated messge selector is applied to the object contained in Register R. The resulting method is evaluated. After evaluation, register R will contain the value returned from the method and the contexts of the X and A register are undefined. Any stacked arguments will have been removed from the stack.

The bytecode encodes the number of arguments except for the common selector bytecodes where the argument count is obtained from the byte vector accessed as known object TDB.

If the message lookup fails, the arguments, receiver, and selector are placed into a Message object. Any stacked arguments are removed from the stack. The message #doesNotUnderstand is sent to the object in the R register.

LRSendDrop is just like a normal message send except that immediately before the send the R register is loaded from the stack location immediately preceding the first argument to the send. For an *x* argument method this is equivalent to having a LRStk *x*-1 bytecode immediate preceding the send bytecode. In addition, upon return from the send the top element of the stack (the same location that was loaded into R) is deleted. This is equivalent to following the send with a DeleteTOS instruction.

Alt	Alternative Semantic Variation Prefix Code E3

The immediately following bytecode is to be interpreted using its alternative semantics. The byte may only prefix the following bytecodes:

SendC	
SendLit	Send to super
LATmp	LXTmp– Load X from Stack Temp
LALit	LXLit– Load X from literal
LAAsc	LXASC– Load X from literal association
LAEnv	LXENV– Load X from environment variable
LAI	LXI– Load X indirect through X
LAConst 0	LRConst 0– Load R with constant 0
LAConst 1	LRConst 1– Load R with constant 1
LAConst 2	LRConst 2– Load R with constant 2
LAConst n	LRConst n– Load R with constant n
RtoA	AtoR – Load R with value from A
LRStk n	LAStk– Load A with Stack value n
SRTmp n	SATmp– Store A into temp n
SREnv n	SAEnv– StoreA into Environment variable n

Ret	Return value of R register Code E0
RetT0	Return value of Temporary 0 Code E1

The designated value is loaded into the R register, if necessary. The current stack frame is discarded including any stacked arguments. The previous stack frame is made the current frame. The SP, FP, and PC registers are restored to their previous values. The contents of the A and X registers are undefined.

RetHome Return From Home Activation
Code E2

Return from the home activation for the current method. For a block this returns from the invocation of the method that created the block. If any protected blocks have been specified between the home method and the current block then the protected blocks are evaluated before the return is done.

Bytecode Encodings

	0-7	8-F
0	Label	Jump
1	JumpT	JumpF
2	loada[1] temp	loadr temp
3	loada[1] lit	loadr lit
4	loada[1] assoc	loadr assoc
5	loada[1] (env)	loadr (env)
6	loada[1] (temp0)	loadr (temp0)
7	loada/r (X) 0-3	store temp (a)
8	store (env) (a)	store (temp0)
9	store (X) 0-3/ loada,r args 1-2	send literal selector[2]
A	send literal selector[2]	send literal selector[2]
B	send common selector[2]	send common selector[2]
C	send common selector[2]	send common selector[2]
D	send common selector[2]	misc
E	misc	misc
F	misc	misc

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D	send	send	send	send	send	send	send	send	a:= 0	a:=1	a:=2		LR TOS	LR 1 TOS	LR 2 TOS2	
E	RET	RET T0	RET Home	alt	POP A	POP R	PUSH R	drop tos	a := r	a:= block	a:=nil	r:=nil	a:= true	r:= true	a:= false	r:= true
F	MPx Label	LTMP	LALI T	LRLiT	LASC	LASC	LENV LI	LRT0	LRT0	X1	STO	SI	send literal	xx send		reserved

X1 second byte encoding:

00+n	Extended store R into stack temp n
40+n	Extended store R indirect through environment pointer
80+n	Load A from stack entry n (0=n=15)
A0+n	Load A from stacked argument n
B0+n	Load R from stacked argument n
C0+n	Load A with excess-32 encoded integer n (-32=n=31)

[1] Escaped to load X register

[2] Escaped to send to Super.

To Do

Interrupt Frames, especially for frameless methods.
Callback frames.